Subject: [PATCH 9/10] BC: physical pages accounting/reclamation (core)
Posted by Kirill Korotaev on Thu, 05 Oct 2006 15:55:40 GMT
View Forum Message <> Reply to Message

Accounting of physical pages.
Accounting is performed on page mapping into address space.
If page is shared, it is charged to its first user.

If physpages limit is hit, BC pages are reclaimed on the
same basis as try_to_free_pages() does.

Full BC implementation allows fractions of pages accounting
as well: http://wiki.openvz.org/RSS_fractions_accounting

Signed-off-by: Pavel Emelianov <xemul@openvz.org>
Signed-off-by: Kirill Korotaev <dev@openvz.org>

```
 include/bc/rsspages.h      |   46 +++++++
 include/linux/mm.h         |    1
 include/linux/mm_types.h   |    5
 include/linux/page-flags.h |    4
 kernel/bc/rsspages.c       |  270 ++++++++++++++++++++++++++++++++++++++++++++++
 mm/vmscan.c                |  190 ++++++++++++++++++++++++++++++++
 6 files changed, 515 insertions(+), 1 deletion(-)

--- ./include/bc/rsspages.h.bc_rsspages 2006-10-05 12:20:42.000000000 +0400
+++ ./include/bc/rsspages.h 2006-10-05 12:35:57.000000000 +0400
@@ -0,0 +1,46 @@
+/*
+ * include/bc/rsspages.h
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#ifndef __BC_RSSPAGES_H_
+#define __BC_RSSPAGES_H_
+
+#include <linux/compiler.h>
+
+struct page;
+struct vm_area_struct;
+struct page_beancounter;
+
+#ifdef CONFIG_BEANCOUNTERS
+int __must_check bc_rsspage_prepare(struct page *p,
```

```
+  struct vm_area_struct *vma, struct page_beancounter **ppb);
+void bc_rsspage_charge(struct page_beancounter *pb);
+void bc_rsspage_release(struct page_beancounter *pb);
+void bc_rsspage_uncharge(struct page *p);
+
+unsigned long bc_try_to_free_pages(struct beancounter *bc);
+unsigned long bc_isolate_pages(unsigned long nr_to_scan,
+  struct beancounter *bc, struct list_head *dst,
+  int active, unsigned long *scanned);
+unsigned long bc_nr_physpages(struct beancounter *bc);
+#else
+static inline int __must_check bc_rsspage_prepare(struct page *p,
+  struct vm_area_struct *vma, struct page_beancounter **ppb)
+{
+ return 0;
+}
+
+static inline void bc_rsspage_charge(struct page_beancounter *pb)
+{
+}
+static inline void bc_rsspage_release(struct page_beancounter *pb)
+{
+}
+static inline void bc_rsspage_uncharge(struct page *p)
+{
+}
+#endif
+#endif
--- ./include/linux/mm.h.bc_rsspages 2006-10-05 12:16:03.000000000 +0400
+++ ./include/linux/mm.h 2006-10-05 12:36:42.000000000 +0400
@@ -223,6 +223,7 @@ struct mmu_gather;
 struct inode;

 #define page_bc(page)   ((page)->bc)
+#define page_pb(page)   ((page)->pb)
 #define page_private(page)  ((page)->private)
 #define set_page_private(page, v) ((page)->private = (v))

--- ./include/linux/mm_types.h.bc_rsspages 2006-10-05 12:12:24.000000000 +0400
+++ ./include/linux/mm_types.h 2006-10-05 12:36:21.000000000 +0400
@@ -63,7 +63,10 @@ struct page {
        not kmapped, ie. highmem) */
 #endif /* WANT_PAGE_VIRTUAL */
 #ifdef CONFIG_BEANCOUNTERS
- struct beancounter *bc;
+ union {
+  struct beancounter *bc;
+  struct page_beancounter *pb;
```

```
+ };
 #endif
 #ifdef CONFIG_PAGE_OWNER
  int order;
--- ./include/linux/page-flags.h.bc_rsspages 2006-10-05 11:42:43.000000000 +0400
+++ ./include/linux/page-flags.h 2006-10-05 12:20:42.000000000 +0400
@@ -92,6 +92,7 @@
 #define PG_buddy  19 /* Page is free, on buddy lists */

 #define PG_readahead  20 /* Reminder to do readahead */
+#define PG_charged  21 /* Page is already charged to BC */


 #if (BITS_PER_LONG > 32)
@@ -253,6 +254,9 @@ static inline void SetPageUptodate(struc
 #define SetPageReadahead(page) set_bit(PG_readahead, &(page)->flags)
 #define TestClearPageReadahead(page) test_and_clear_bit(PG_readahead, &(page)->flags)

+#define TestSetPageCharged(page) test_and_set_bit(PG_charged, &(page)->flags)
+#define ClearPageCharged(page) clear_bit(PG_charged, &(page)->flags)
+
 struct page; /* forward declaration */

 int test_clear_page_dirty(struct page *page);
--- ./kernel/bc/rsspages.c.bc_rsspages 2006-10-05 12:20:42.000000000 +0400
+++ ./kernel/bc/rsspages.c 2006-10-05 12:37:40.000000000 +0400
@@ -0,0 +1,270 @@
+/*
+ * kernel/bc/rsspages.c
+ *
+ * Copyright (C) 2006 OpenVZ SWsoft Inc
+ *
+ */
+
+#include <linux/mm_types.h>
+#include <linux/mm.h>
+#include <linux/page-flags.h>
+#include <linux/hardirq.h>
+#include <linux/kernel.h>
+
+#include <bc/beancounter.h>
+#include <bc/vmpages.h>
+#include <bc/rsspages.h>
+
+#include <asm/bitops.h>
+
+#define BC_PHYSPAGES_BARRIER BC_MAXVALUE
+#define BC_PHYSPAGES_LIMIT BC_MAXVALUE
```

```
+
+/*
+ * page_beancounter is a tie between page and beancounter page is
+ * charged to. it is used to reclaim pages faster by walking bc's
+ * page list, not zones' ones
+ *
+ * this tie can also be used to implement fractions accounting mechanism
+ * as it is done in OpenVZ kernels
+ */
+struct page_beancounter {
+ struct page *page;
+ struct beancounter *bc;
+ struct list_head list;
+};
+
+/*
+ * API calls
+ */
+
+/*
+ * bc_rsspage_prepare allocates a tie and charges page to vma's beancounter
+ * this must be called in non-atomic context to give a chance for pages
+ * reclaiming. otherwise hitting limits will cause -ENOMEM returned.
+ */
+int bc_rsspage_prepare(struct page *page, struct vm_area_struct *vma,
+  struct page_beancounter **ppb)
+{
+ struct beancounter *bc;
+ struct page_beancounter *pb;
+
+ pb = kmalloc(sizeof(struct page_beancounter), GFP_KERNEL);
+ if (pb == NULL)
+  goto out_nomem;
+
+ bc = vma->vma_bc;
+ if (bc_charge(bc, BC_PHYSPAGES, 1, BC_LIMIT))
+  goto out_charge;
+
+ pb->page = page;
+ pb->bc = bc;
+ *ppb = pb;
+ return 0;
+
+out_charge:
+ kfree(pb);
+out_nomem:
+ return -ENOMEM;
+}
```

```
+
+/*
+ * bc_rsspage_release is a rollback call for bc_rsspage_prepare
+ */
+void bc_rsspage_release(struct page_beancounter *pb)
+{
+ bc_uncharge(pb->bc, BC_PHYSPAGES, 1);
+ kfree(pb);
+}
+
+/*
+ * bc_rsspage_charge actually ties page and beancounter together
+ * and marks page as PG_charged. this is done in not-failing path
+ * to be sure the page IS charged
+ */
+void bc_rsspage_charge(struct page_beancounter *pb)
+{
+ struct page *pg;
+ struct beancounter *bc;
+
+ pg = pb->page;
+ if (TestSetPageCharged(pg)) {
+  bc_rsspage_release(pb);
+  return;
+ }
+
+ bc = bc_get(pb->bc);
+ spin_lock(&bc->bc_page_lock);
+ list_add(&pb->list, &bc->bc_page_list);
+ spin_unlock(&bc->bc_page_lock);
+ page_pb(pg) = pb;
+}
+
+/*
+ * bc_rsspage_uncharge is called when pages is get completely unapped
+ * from all address spaces
+ */
+void bc_rsspage_uncharge(struct page *page)
+{
+ struct page_beancounter *pb;
+ struct beancounter *bc;
+
+ pb = page_pb(page);
+ if (pb == NULL)
+  return;
+
+ ClearPageCharged(page);
+ page_pb(page) = NULL;
```

```
+
+ bc = pb->bc;
+ spin_lock(&bc->bc_page_lock);
+ list_del(&pb->list);
+ spin_unlock(&bc->bc_page_lock);
+
+ bc_uncharge(bc, BC_PHYSPAGES, 1);
+ bc_put(bc);
+ kfree(pb);
+}
+
+/*
+ * Page reclamation helper
+ *
+ * this function resembles isolate_lru_pages() but is scans through
+ * bc's page list, not zone's active/inactive ones.
+ */
+
+unsigned long bc_isolate_pages(unsigned long nr_to_scan, struct beancounter *bc,
+  struct list_head *dst, int active, unsigned long *scanned)
+{
+ unsigned long nr_taken = 0;
+ struct page *page;
+ struct page_beancounter *pb;
+ unsigned long scan;
+ struct list_head *src;
+ LIST_HEAD(pb_list);
+ struct zone *z;
+
+ spin_lock(&bc->bc_page_lock);
+ src = &bc->bc_page_list;
+ for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
+  struct list_head *target;
+  pb = list_entry(src->prev, struct page_beancounter, list);
+  page = pb->page;
+  z = page_zone(page);
+
+  list_move(&pb->list, &pb_list);
+
+  spin_lock_irq(&z->lru_lock);
+  if (PageLRU(page)) {
+   if ((active && PageActive(page)) ||
+      (!active && !PageActive(page))) {
+    if (likely(get_page_unless_zero(page))) {
+     ClearPageLRU(page);
+     target = dst;
+     nr_taken++;
+     list_move(&page->lru, dst);
```

```
+    }
+   }
+  }
+  spin_unlock_irq(&z->lru_lock);
+ }
+
+ list_splice(&pb_list, src);
+ spin_unlock(&bc->bc_page_lock);
+
+ *scanned = scan;
+ return nr_taken;
+}
+
+unsigned long bc_nr_physpages(struct beancounter *bc)
+{
+ return bc->bc_parms[BC_PHYSPAGES].held;
+}
+
+/*
+ * Generic resource info
+ */
+
+static int bc_phys_init(struct beancounter *bc, gfp_t mask)
+{
+ spin_lock_init(&bc->bc_page_lock);
+ INIT_LIST_HEAD(&bc->bc_page_list);
+
+ bc_init_resource(bc, BC_PHYSPAGES,
+   BC_PHYSPAGES_BARRIER, BC_PHYSPAGES_LIMIT);
+ return 0;
+}
+
+static void bc_phys_barrier_hit(struct beancounter *bc)
+{
+ /*
+  * May wake up kswapd here to start asynchronous reclaiming of pages
+  */
+}
+
+static int bc_phys_limit_hit(struct beancounter *bc, unsigned long val,
+  unsigned long flags)
+{
+ int did_some_progress = 0;
+ struct bc_resource_parm *parm;
+
+ spin_unlock_irqrestore(&bc->bc_lock, flags);
+ might_sleep();
+
```

```
+ parm = &bc->bc_parms[BC_PHYSPAGES];
+ while (1) {
+  did_some_progress = bc_try_to_free_pages(bc);
+
+  spin_lock_irq(&bc->bc_lock);
+  if (parm->held + val <= parm->limit) {
+   parm->held += val;
+   bc_adjust_maxheld(parm);
+   return 0;
+  }
+
+  if (!did_some_progress) {
+   parm->failcnt++;
+   return -ENOMEM;
+  }
+  spin_unlock_irq(&bc->bc_lock);
+ }
+}
+
+static int bc_phys_change(struct beancounter *bc,
+  unsigned long barrier, unsigned long limit)
+{
+ int did_some_progress;
+ struct bc_resource_parm *parm;
+
+ parm = &bc->bc_parms[BC_PHYSPAGES];
+ if (limit >= parm->held)
+  return 0;
+
+ while (1) {
+  spin_unlock_irq(&bc->bc_lock);
+
+  did_some_progress = bc_try_to_free_pages(bc);
+
+  spin_lock_irq(&bc->bc_lock);
+  if (parm->held < limit)
+   return 0;
+  if (!did_some_progress)
+   return -ENOMEM;
+ }
+}
+
+struct bc_resource bc_phys_resource = {
+ .bcr_name  = "physpages",
+ .bcr_init  = bc_phys_init,
+ .bcr_change  = bc_phys_change,
+ .bcr_barrier_hit = bc_phys_barrier_hit,
+ .bcr_limit_hit  = bc_phys_limit_hit,
```

```
+};
+
+static int __init bc_phys_init_resource(void)
+{
+ bc_register_resource(BC_PHYSPAGES, &bc_phys_resource);
+ return 0;
+}
+
+__initcall(bc_phys_init_resource);
--- ./mm/vmscan.c.bc_rsspages 2006-10-05 11:42:43.000000000 +0400
+++ ./mm/vmscan.c 2006-10-05 12:33:06.000000000 +0400
@@ -38,6 +38,8 @@
 #include <linux/delay.h>
 #include <linux/kthread.h>

+#include <bc/rsspages.h>
+
 #include <asm/tlbflush.h>
 #include <asm/div64.h>

@@ -1076,6 +1078,194 @@ out:
  return ret;
 }

+#ifdef CONFIG_BEANCOUNTERS
+/*
+ * These are bc's inactive and active pages shrinkers.
+ * Thes works like shrink_inactive_list() and shrink_active_list()
+ *
+ * Two main differences is that bc_isolate_pages() is used to isolate
+ * pages, and that reclaim_mapped is considered to be 1 as hitting BC
+ * limit implies we have to shrink _mapped_ pages
+ */
+static unsigned long bc_shrink_pages_inactive(unsigned long max_scan,
+  struct beancounter *bc, struct scan_control *sc)
+{
+ LIST_HEAD(page_list);
+ unsigned long nr_scanned = 0;
+ unsigned long nr_reclaimed = 0;
+
+ do {
+  struct page *page;
+  unsigned long nr_taken;
+  unsigned long nr_scan;
+  struct zone *z;
+
+  nr_taken = bc_isolate_pages(sc->swap_cluster_max, bc,
+    &page_list, 0, &nr_scan);
```

```
+
+  nr_scanned += nr_scan;
+  nr_reclaimed += shrink_page_list(&page_list, sc);
+  if (nr_taken == 0)
+   goto done;
+
+  while (!list_empty(&page_list)) {
+   page = lru_to_page(&page_list);
+   z = page_zone(page);
+
+   spin_lock_irq(&z->lru_lock);
+   VM_BUG_ON(PageLRU(page));
+   SetPageLRU(page);
+   list_del(&page->lru);
+   if (PageActive(page))
+    add_page_to_active_list(z, page);
+   else
+    add_page_to_inactive_list(z, page);
+   spin_unlock_irq(&z->lru_lock);
+
+   put_page(page);
+  }
+   } while (nr_scanned < max_scan);
+done:
+ return nr_reclaimed;
+}
+
+static void bc_shrink_pages_active(unsigned long nr_pages,
+  struct beancounter *bc, struct scan_control *sc)
+{
+ LIST_HEAD(l_hold);
+ LIST_HEAD(l_inactive);
+ LIST_HEAD(l_active);
+ struct page *page;
+ unsigned long nr_scanned;
+ unsigned long nr_deactivated = 0;
+ struct zone *z;
+
+ bc_isolate_pages(nr_pages, bc, &l_hold, 1, &nr_scanned);
+
+ while (!list_empty(&l_hold)) {
+  cond_resched();
+  page = lru_to_page(&l_hold);
+  list_del(&page->lru);
+  if (page_mapped(page)) {
+   if ((total_swap_pages == 0 && PageAnon(page)) ||
+       page_referenced(page, 0)) {
+    list_add(&page->lru, &l_active);
```

```
+    continue;
+   }
+  }
+  nr_deactivated++;
+  list_add(&page->lru, &l_inactive);
+ }
+
+ while (!list_empty(&l_inactive)) {
+  page = lru_to_page(&l_inactive);
+  z = page_zone(page);
+
+  spin_lock_irq(&z->lru_lock);
+  VM_BUG_ON(PageLRU(page));
+  SetPageLRU(page);
+  VM_BUG_ON(!PageActive(page));
+  ClearPageActive(page);
+
+  list_move(&page->lru, &z->inactive_list);
+  z->nr_inactive++;
+  spin_unlock_irq(&z->lru_lock);
+
+  put_page(page);
+ }
+
+ while (!list_empty(&l_active)) {
+  page = lru_to_page(&l_active);
+  z = page_zone(page);
+
+  spin_lock_irq(&z->lru_lock);
+  VM_BUG_ON(PageLRU(page));
+  SetPageLRU(page);
+  VM_BUG_ON(!PageActive(page));
+  list_move(&page->lru, &z->active_list);
+  z->nr_active++;
+  spin_unlock_irq(&z->lru_lock);
+
+  put_page(page);
+ }
+}
+
+/*
+ * This is a reworked shrink_zone() routine - it scans active pages firts,
+ * then inactive and returns the number of pages reclaimed
+ */
+static unsigned long bc_shrink_pages(int priority, struct beancounter *bc,
+  struct scan_control *sc)
+{
+ unsigned long nr_pages;
```

```
+ unsigned long nr_to_scan;
+ unsigned long nr_reclaimed = 0;
+
+ nr_pages = (bc_nr_physpages(bc) >> priority) + 1;
+ if (nr_pages < sc->swap_cluster_max)
+  nr_pages = 0;
+
+ while (nr_pages) {
+  nr_to_scan = min(nr_pages, (unsigned long)sc->swap_cluster_max);
+  nr_pages -= nr_to_scan;
+  bc_shrink_pages_active(nr_to_scan, bc, sc);
+ }
+
+ nr_pages = (bc_nr_physpages(bc) >> priority) + 1;
+ if (nr_pages < sc->swap_cluster_max)
+  nr_pages = 0;
+
+ while (nr_pages) {
+  nr_to_scan = min(nr_pages, (unsigned long)sc->swap_cluster_max);
+  nr_pages -= nr_to_scan;
+  nr_reclaimed += bc_shrink_pages_inactive(nr_to_scan, bc, sc);
+ }
+
+ throttle_vm_writeout();
+ return nr_reclaimed;
+}
+
+/*
+ * This functions works like try_to_free_pages() - it tries
+ * to shrink bc's pages with increasing priority
+ */
+unsigned long bc_try_to_free_pages(struct beancounter *bc)
+{
+ int priority;
+ int ret = 0;
+ unsigned long total_scanned = 0;
+ unsigned long nr_reclaimed = 0;
+ struct scan_control sc = {
+  .gfp_mask = GFP_KERNEL,
+  .may_writepage = !laptop_mode,
+  .swap_cluster_max = SWAP_CLUSTER_MAX,
+  .may_swap = 1,
+  .swappiness = vm_swappiness,
+ };
+
+ for (priority = DEF_PRIORITY; priority >= 0; priority--) {
+  sc.nr_scanned = 0;
+  nr_reclaimed += bc_shrink_pages(priority, bc, &sc);
```

```
+  total_scanned += sc.nr_scanned;
+  if (nr_reclaimed >= sc.swap_cluster_max) {
+   ret = 1;
+   goto out;
+  }
+
+  if (total_scanned > sc.swap_cluster_max +
+     sc.swap_cluster_max / 2) {
+   wakeup_pdflush(laptop_mode ? 0 : total_scanned);
+   sc.may_writepage = 1;
+  }
+
+  if (sc.nr_scanned && priority < DEF_PRIORITY - 2)
+   blk_congestion_wait(WRITE, HZ/10);
+ }
+out:
+ return ret;
+}
+#endif
+
 /*
  * For kswapd, balance_pgdat() will work across all this node's zones until
  * they are all at pages_high.
```